

# Chapter 5

## Spatial Data-structures for point set surfaces

In this chapter, we discuss how to utilize spatial data-structures in order to accelerate the computation of point-based surface representations. In this context, spatial data-structures can be applied for the following purposes:

1. **Evaluation of the surface:** The implicit function  $f$  can be evaluated locally due to the usage of compactly supported weighting function. Acceleration hierarchies are used to quickly find the relevant points that contribute to the evaluation of the surface.
2. **Finding potential ray-surface intersections:** This task is performed to provide the ray surface intersection algorithm with initial locations to start the iterative procedure.

The first task has to be performed every time, when evaluating the surface. The latter task is usually performed during rendering, but the computation of ray-surface intersections can also be necessary to implement certain shape modelling operations. Here, general techniques for accelerating *ray-shooting algorithms* (also called *ray traversal algorithms*) apply. An overview on this field can be found in [52, 97]. Hierarchies which have been used in the context of point-based surface representations are *octrees* [2], *bounding sphere hierarchies* [3, 1] and *kd-trees* [98].

Following, we describe how to efficiently evaluate the surface (Section 5.1) and how to quickly determine potential ray-surface intersections (Section 5.2). Finally, we propose how to integrate both concerns into one data-structure (Section 5.3). This includes examples of optimized acceleration hierarchies for static and deforming point models.

## 5.1 Evaluation of the surface

In order to evaluate the implicit function  $f(\mathbf{x})$ , we need to find all the relevant points  $\mathbf{p}_i$ , for which the weighting function  $\theta$  does not evaluate to zero, i.e.

$$\mathcal{P}_S(\mathbf{x}) = \{p_i \in \mathcal{P} | \theta_i(\|\mathbf{x} - \mathbf{p}_i\|) > 0\}. \quad (5.1)$$

This usually has to be done for every intermediate step of the projection procedure or the ray-surface intersection algorithm, which both are iterative processes. Considering all the points is not feasible, as point based models often consist of thousands or millions of points.

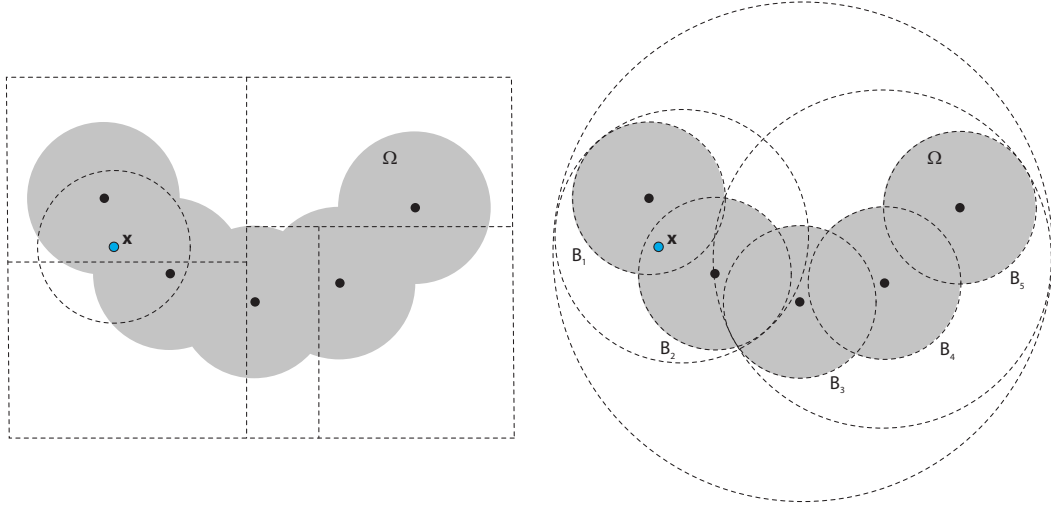
Instead, for compactly supported weighting functions, it is possible to construct bounding volumes that completely contain this area. With these, standard spatial data-structures can be applied, which allow to quickly determine  $\mathcal{P}_S(\mathbf{x})$  by performing range queries or containment testing. For bounding radially symmetric weighting functions balls  $B_i$  can be chosen. In this case, the supported area  $\Omega$  is described exactly. In other cases, e.g. when the supported region is ellipsoidal, other bounding volumes can be used. It can be advantageous to use conservative bounding volumes, although some of the returned points have no contribution. Such examples will be discussed further below.

We describe two possible ways to apply spatial data structures in order to efficiently find the set of contributing points. The first way is to perform an in radius search on points. This is only applicable for compactly supported weight-functions with equal radius of support for all sample points. The second and more general way is to use bounding volumes for the samples and to determine all the volumes that contain a point in space. This is also applicable for weight functions where the extent varies from sample to sample, which is the case in feature adaptive representations, discussed in Chapter 8.

### 5.1.1 In Radius Search on Points

When dealing with uniformly sampled models, it makes sense to calculate the contribution of each sample with the same radially symmetric weighting function. This is the basic approach of calculating  $f(\mathbf{x})$ , as described in Section 2.4. For compactly supported weighting functions, all we have to do is to find the points around  $\mathbf{x}$  within the radius of support. This kind of query often is referred to as "in radius search".

Typically, kd-Trees are used to perform such queries. For points they can be quickly generated and they consume very little memory, thus allowing computationally efficient queries. In principle, during traversal it is necessary to compare the query ball to the extents of each subtree, which corresponds to a sphere-box intersection test (see Figure 5.1, left). These are very simple because the boxes enclosing the subnodes are



**Figure 5.1:** Left: The samples are arranged in a kd-tree. Searching the  $\mathcal{P}_S$  amounts to performing an in-radius-query of radius  $h$ . Right: The weighting functions are bounded by volumes  $B_i$  which are arranged in spatial data-structure. This approach can handle  $B_i$  of varying size. However, querying  $\mathcal{P}_S$  is more involved.

axis-aligned. The extents are computed on the fly, as keeping memory compact over-compensates the cost for the extra computation.

### 5.1.2 Using bounding volumes for the weighting functions

For non-uniform samplings, as described later, in Chapter 8, finding the points that contribute to the evaluation of  $f$  at  $\mathbf{x}$  is more complex. In this case, different weight functions<sup>1</sup> are used for each sample. In order to make use of spatial data-structures to efficiently determine the relevant points  $\mathcal{P}_S(\mathbf{x})$ , we need to construct a bounding volume for each sample that contains the supported area (see Figure 5.1, right). Then, we have to find all the bounding volumes that contain the point  $x$ , where  $f$  is evaluated. The in radius search described in the previous section can be seen as a special case of this, where all weighting functions have the same radii. Then, looking for points within a ball of radius  $r$  around  $\mathbf{x}$  is equivalent to searching the balls of radius  $r$  that contain  $\mathbf{x}$ .

The bounding volumes can also be chosen conservatively. For instance, instead of using ellipsoids to bound ellipsoidal weighting functions, oriented bounding boxes (OBBs) can be used. This might lead to more efficient queries due to cheaper inside/outside-tests, although the contributions of some of the found points evaluate to zero.

<sup>1</sup>At least the radius of support varies!

In order to quickly find the samples  $\mathcal{P}_S(\mathbf{x})$ , we construct a spatial data-structure from the corresponding bounding volumes. Both bounding volume hierarchies and space partitioning strategies can be applied to determine the bounding volumes containing the query point. For static models, space partitioning is preferable, as the queries can be performed more efficiently. For dynamic models, bounding volume hierarchies should be used, as the volumes can be adapted, not requiring to rebuild the tree-structure after every change. Compared to the in radius search on points, here, the spatial data-structures require more memory, as the objects stored are volumetric. This implies that they can either fall into multiple leafs, when using space partitioning or the bounding volumes of the inner nodes overlap, when using bounding volume hierarchies. In addition, building efficient spacial data-structures on volumetric objects is more time consuming, as more complex containment testing is necessary. Nevertheless, for static models arranged in a space partitioning querying is usually faster, as the points  $\mathcal{P}_S(\mathbf{x})$  can be stored in the leaf cells a-priori, which only requires to find the leaf node containing the query point. In addition, follow-up queries can be done without repeating the queries, as long as the query point is within the same voxel. This often applies when projecting or traversing a ray.

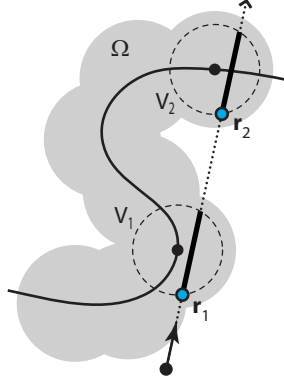
## 5.2 Finding Potential Ray-surface Intersections

During ray-tracing or ray-casting, ray-surface intersection computation has to be performed. The algorithms described in Chapter 4.2 require to find an initial point close to the surface (see Figure 5.2). Moreover, this initial point has to be near the first potential intersection of the ray and the surface. If no intersection occurs, the next potential point of intersection has to be examined.

For implicit surfaces the extents of  $S$  are not known a priori. Therefore, we conservatively enclose the surface in bounding volumes and arrange them in a spatial data-structure. The intersection of the ray with the bounding volume is used as initial point for the ray-surface intersection algorithm. Together with the point where the ray leaves the bounding volume, a ray segment potentially containing the ray-surface intersection is given.

For performance reasons it is important to enclose the entire surface as tightly as possible. Tighter bounding volumes reduce the number of rays to be examined, which each time results in the invocation of the ray-surface intersection algorithm. In addition, the extents of the relevant ray segment becomes shorter.

It must also be possible to evaluate  $f$  on the entire ray segment without running into numerical problems that arise from extremely small weights, i.e. the sum of weights in Equation 2.1 has to be significantly larger than zero everywhere. Therefore, the union of the bounding volumes enclosing the surface has to be completely contained in  $\Omega$ . We



**Figure 5.2:** In order to determine the initial points  $\mathbf{r}_i$  and the ray segments to be examined, the ray is clipped against the  $V_i$ . As the  $V_i$  are completely contained in  $\omega$ , there is support on the entire ray-segments.

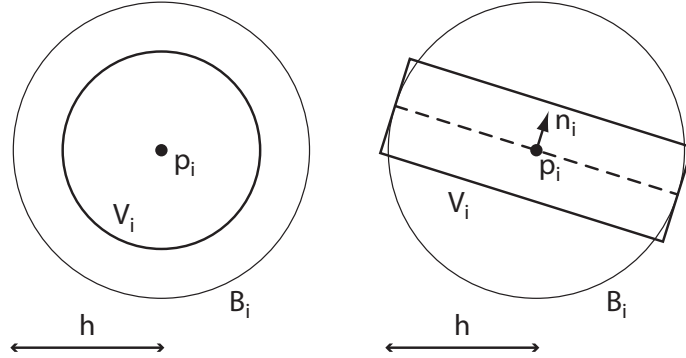
denote the bounding volumes enclosing a surface patch by  $V_i$  and their union by  $\mathcal{V}$ .

The ray-surface intersection algorithms presented in Section 4.2 also require a termination criterion, which is given with the bounding volumes: if the current position on the ray leaves the considered ray segment, the evaluation is stopped. The next relevant segment has to be inspected. The bounding volumes have to ensure that the nearest intersection is found. Several potential intersections within the same bounding volume should be avoided. This means we have to separate different patches of the surface. As the rays can have arbitrary directions, the situation can generally not be resolved at silhouettes. When the ray-surface intersection algorithm only examines the ray segment in a forward manner, no problems arise. Otherwise, we have to pay attention. Particularly, when using the approximating and intersecting strategy, we have to ensure that the first intersection is found. This is addressed in Section 3.2.2.

### 5.2.1 Enclosing the surface

In order to enclose the surface, we construct a bounding volume around each sample  $\mathbf{p}_i$ , as we expect the surface to be close to the input points. We propose two types of volumes  $V_i$ : spheres and oriented bounding boxes.

Spheres can be used without taking normals into consideration. We down-scale the  $V_i$  to an amount such that neighboring spheres still overlap to extent where no holes arise (see Figure 5.3, left). This can be done by collecting the  $k$  nearest neighbors. As they are close in space, they are also close on the surface. Irregular samplings can be fixed by symmetrizing the  $k$ -nearest neighbor graph. The distance to the farthest point among the neighbors yields the radius of  $V_i$ . A  $k$  of 6 – 10, depending on the regularity



**Figure 5.3:** Different choices for the  $V_i$ , which contain a patch of  $\mathcal{S}$ . The union of these bounding volumes enclose  $\mathcal{S}$ . Left: the support radius is down-scaled. Right: a bounding box can be used if normals are provided. A plane normal to  $\mathbf{n}_i$  through  $p_i$  is supposed to approximate  $\mathcal{S}$  close to  $p_i$ . Therefore, a sufficiently small box which is down-scaled in normal direction still encloses  $\mathcal{S}$ .

of the sampling, produces good results.

The result is a tighter and more effective bounding volume for  $\mathcal{S}$  that is contained in  $\Omega$ . Moreover, at intersection points with the  $V_i$ , the weight functions now have sufficient support and  $f$  can be evaluated without running into numerical problems.

If normals are provided with the samples, it is also possible to use oriented bounding boxes (see Figure 5.3, right). As  $\mathcal{S}$  is expected to be close to the planes formed by the samples and their normals, the boxes can be down-scaled (e.g. factor  $\frac{1}{5}$  of its radius) in normal direction. The union of the boxes yields an even smaller bounding volume for  $\mathcal{S}$ .

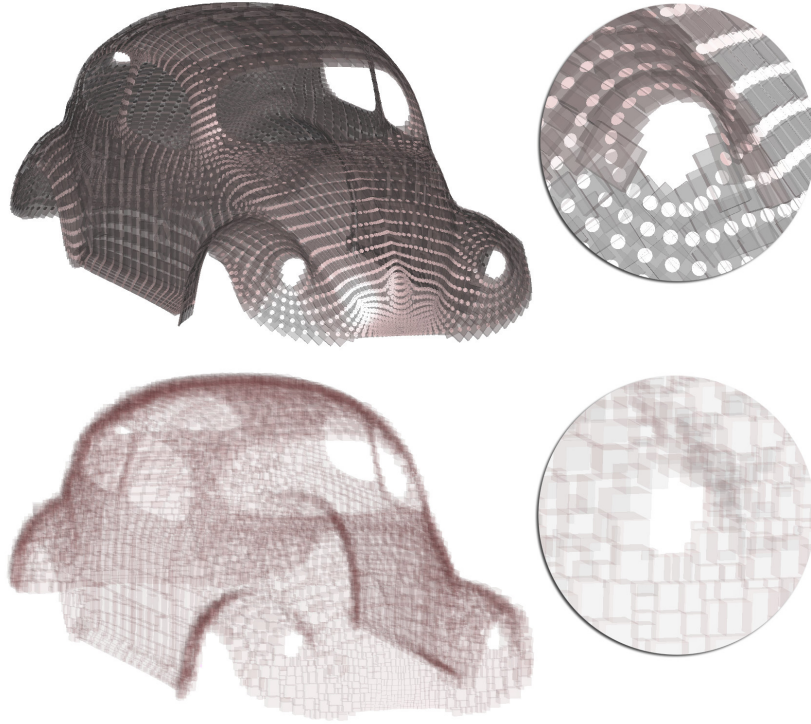
For samples equipped with normals, the conditions for obtaining a hole free covering are similar to the conditions desired for surface splatting. In that context, Wu and Kobbelt [104] propose an algorithm to construct splats from a dense point cloud which satisfy the aforementioned conditions. If adequate surfels are available, they can be expanded in normal direction, to ensure all of the surface is contained. Nevertheless, instead of dealing with such cylinder caps, we propose to use oriented bounding boxes for efficient containment and intersection testing, instead.

The intersections of the ray and the  $V_i$  also serve as region of trust: if the ray-surface algorithm leaves the ray segment defined by the intersections the next  $V_i$  has to be examined.

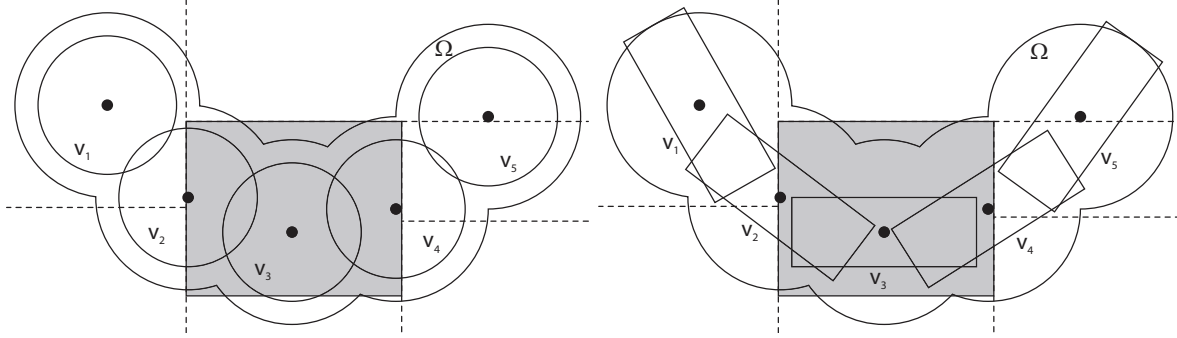
### 5.2.2 Considerations for building the tree

When using space partitioning (e.g. kd-trees), the ray traversal through the consecutive voxels can be performed very efficiently. If all non-empty voxels are completely contained in  $\Omega$ , the intersections of the ray and a voxel can be used as the initial points for the ray-surface intersection algorithm. Intersecting the  $\mathcal{V}_i$  can be avoided. However, this usually requires to build deep trees that are sufficiently tight.

Optionally, the relevant ray segment can be cropped to the extents of the  $V_i$ , before testing for ray-surface intersections. This still ensures that  $f$  can be evaluated on the whole ray-segment and the requirements for the kd-tree are not as demanding.



**Figure 5.4:** The beetle model represented by surfels. Their OBBs (top) are down-scaled in normal direction and used to build a *kd*-tree (bottom). The non-empty voxels are completely contained inside  $\Omega$



**Figure 5.5:** Two choices for a single kd-tree addressing both ray traversal and surface evaluation: bounding spheres (left) and oriented bounding boxes (right) enclose the surface. The potentially contributing point samples are stored in the leaf-nodes of the kd-tree. Here, the tree does not tightly enclose  $\mathcal{S}$ , thus the voxel is not completely contained in  $\Omega$ .

### 5.3 Combining both Issues

So far, we have described spatial data-structures for the evaluation of the surface and for finding potential ray-surface intersections. In most applications, we want to perform both kind of operations. Therefore, it is desirable to integrate all requirements in a single data-structure. This reduces the memory overhead, and thus increases computational performance.

Following, we describe our choice to integrate the concerns in a single data-structure. This is done for the settings of:

- **Static models:** Here, highly optimized data-structures can be pre-computed which only aim at quickly interrogating the surface; We assume that sufficient computational time is available for the preprocess.
- **Deforming surfaces:** Here, it is important to simultaneously adapt the data-structure to changes of the input points, e.g. during physical simulation or modelling. Less strategies for optimizing fast surface interrogations are applicable.

#### 5.3.1 Framework for static point clouds

Here we consider static point clouds, i.e. only rigid movements are allowed. In that setting, we advocate the use of kd-trees, as tree traversal can be performed very efficiently. (For an comparison of tree traversal algorithms see [52]) How to construct efficient kd-trees for ray shooting is detailed in [97]. The construction basically consists of:



1. **Finding potential splitting locations:** Potential splitting locations along the best splitting dimension have to be identified. This can be done in a draft and a precise way: the first considers the locations of the extents of the bounding box, only; the latter, also takes the current voxel into consideration, which requires complex bounding box to voxel intersection calculations.
2. **Choosing the best splitting location:** The optimal splitting location, which maximizes the empty region, has to be chosen among the aforementioned candidates. This is done according to the Surface Area Heuristic (SAH).

The resulting tree has both empty and non-empty leaves, where the latter preferably are located high up in the tree. This allows to quickly detect rays missing the kd-tree, which minimizes the traversal cost.

We build the tree from the  $V_i$ . Preferably, the tree is sufficiently deep to be contained in  $\Omega$ . In that case, the ray-segment to be examined by the ray-surface intersection procedure (as introduced in Section 3.2) results from intersecting the ray with the current voxel. There are also situations, when the user does not want to spend the time necessary to construct very deep trees (e.g. for previewing or quickly inspecting the model). In that case, the draft splitting locations can be used to build the tree.

As the non-empty voxels usually are not tight enough to completely be contained inside  $\Omega$ , ray-segment, resulting from intersecting with the current voxel, additionally has to be clipped against  $\mathcal{V}$ . This way we guarantee that there is support on the entire ray segment.

After building the tree enclosing  $\mathcal{V}$ , we consider all the non-empty leaf nodes and store references to all the  $\mathbf{p}_i$  for which the  $V_i$  are intersected by the corresponding voxels. This can be done during building the tree and it only requires cheap intersections tests, as the splitting planes are axis aligned. Because we are dealing with volumetric objects, the  $V_i$  can be contained in both children of an inner node. During the build, memory has to be allocated. In order to obtain a compact memory layout, the resulting tree should be copied in a postprocess.

The advantages from storing the  $\mathbf{p}_i$  in the leaf nodes are:

1. **Fast look-up:** To evaluate  $f(\mathbf{x})$ , it is only necessary to determine the voxel that contains  $\mathbf{x}$  and then loop over the contained samples. The samples that do not contribute are simply ignored.
2. **Cheap follow-up evaluations:** Following evaluations inside the same voxel do not require to repeat the look-up. The same set of samples can simply be used again.

Using this construction scheme, we obtain a tightly fitting spatial data-structure which is capable of being efficiently traversed. In addition, it quickly provides the

potentially contributing sample points for evaluating the surface. It is also applicable for weighting functions that differ from sample to sample, which is of importance in feature adaptive representations (for details see 8).

Wald and Seidel [98] use a similar approach. They make use of our surface definition and introduce various optimizations to achieve interactive frame-rates. The performance is basically achieved due to an highly optimized kd-tree acceleration structure which is constructed similarly to the way described above. The non-empty cells of the kd-tree are completely contained in  $\Omega$ . The ray-segment which has to be examined for the current voxel is sampled according to the Ray Marching with linear intersection interpolation method described in Section 3.2. For this, single instruction multiple data (SIMD) techniques are exploited, which allows to evaluate several ray locations during a single loop over the contributing samples.

To further tighten the non-empty kd-cells, they apply the following procedure:

- **The voxels are sliced into sub-cells:** The average normal direction of the contained  $p_i$  is computed and used to construct a stack of equidistant slabs which are normal to that direction.
- **Each sub-cell is sampled randomly:** This way, the minimum and maximum implicit value is estimated.
- **Removal of empty sub-cells:** Only if these values have differing signs, the surface is expected to be contained in the sub-cell. All the others are removed.

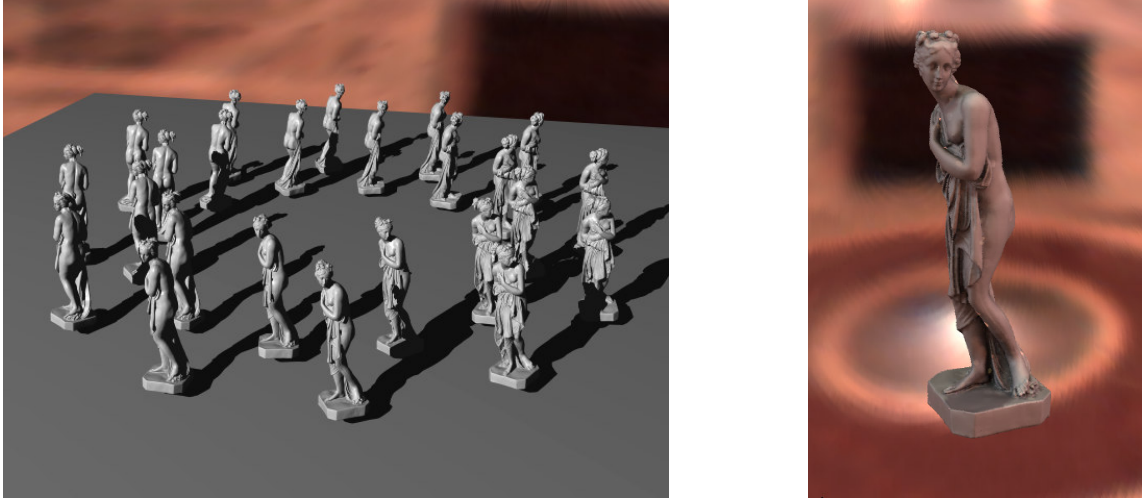
Removing sub-cells enables the split planes of the kd-Tree to be moved towards the surface. They report that 7-13 slices and 100-200 samples per cell work well in practice.

Interactive frame-rates of 7 to 30 frames per second at 512x512 pixels image resolution are obtained (on a single PC) for models of over a million of points. Two scenes rendered interactively are shown in Figure 5.6.

The described optimizations yield high performance during rendering and projecting. Nevertheless, they are only applicable if arbitrary time can be spend on the preprocess that produces the spatial data-structure. Some of the optimizations are not always appropriate. For instance, in the setting of inspecting point based models during or after scanning, the system has to provide the user with previews relatively quickly. This means that only some of the optimizations are affordable. Even more respond-sensitive systems have to be implemented in order to deal with deforming surfaces.

### 5.3.2 Framework for Deforming Surfaces

In this section, we focus on deforming surfaces, such as used for animations, physically based simulation or interactive shape modelling. As stated before, we only want to maintain a single hierarchy. The bounding volumes enclosing the surface are completely contained inside  $B$ . Therefore, we have two choices to build the data-structures:



**Figure 5.6:** Left: a scene consisting of 24 Iphigenias with a total of 24 million points with phong shading and shadows rendered at 2 frames per second at  $640 \times 480$  image resolution. Right: One Iphigenia rendered with precomputed global illumination at 4 frames per second for a resolution of  $400 \times 600$  pixels. Images taken from [98].

- **Based on the  $B_i$ :** This results in a hierarchy that is not very tight. In the leaf nodes we have to store the Bounding volumes enclosing the surface, in order to obtain good initial locations for starting the ray-surface intersection algorithm.
- **From the Bounding volumes enclosing the surface:** To be able to still determine all contributing point samples, we have to compute them separately for each leaf node and store them for later access.

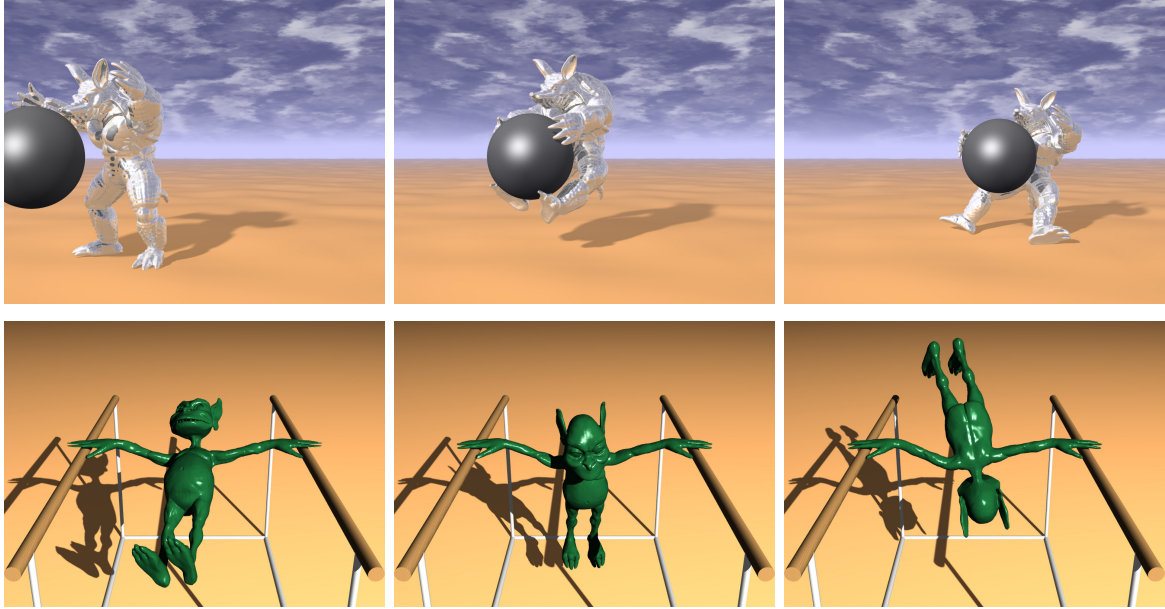
Usually the first approach has to be chosen. There are also situations, where the latter approach can be applied, which is discussed further below.

We have presented a framework [3] based on a bounding sphere hierarchy, which is capable of dealing with deforming surfaces. The hierarchy is built around the  $B_i$  and in the leaf nodes we simply use a global scaling factor to enclose the surface. No extra memory has to be spent on storing the bounding volumes enclosing the surface. During ray intersection, many rays are not rejected until reaching the leaf nodes. However, intersection testing of rays and spheres is a rather cheap operation.

The hierarchy is built top-down starting with a sphere wrapped around all point samples. Each sphere is recursively split into two child spheres until eventually a sphere contains only one single sample. Splitting is done according to the plane defined by the longest axis of the samples' bounding box, similar to [87]. The sphere centers and optimal radii are computed using the miniball algorithm of [42].

The main advantage of this acceleration structure is its ability to quickly adapt to changes, when the model is altered. The hierarchy is updated in a lazy manner, i.e. the radii of the parents of all affected leaf nodes are recalculated. This is much cheaper than rebuilding the tree and can be done during user interaction. When resources are available, the tree structure can either be optimized partially or it can be completely rebuild.

In order to quickly find an initial intersection point for the iterative intersection algorithm, we start intersecting the parent’s bounding sphere. If it is hit, we proceed with both children. This can then be repeated recursively until the ray hits a leaf sphere and the iterative intersection algorithm is invoked.



**Figure 5.7:** Top row: three frames from the cannon ball armadillo sequence. The armadillo is defined by 170k surface points and is animated using 453 simulation nodes. Bottom row: three frames from the gymnastic goblin sequence. The goblin is defined by 100k surface points and is animated using 502 simulation nodes. Images taken from [1].

Collecting the contributing samples is done by testing if  $\mathbf{x}$  is contained in the bounding spheres. We start at the parent node and traverse all children for which the test succeeds. At the leaf node level, we collect the samples.

We are able to reuse the set of collected points if the relevant range of locations to evaluate is known a priori. This requires slightly more complex containment- or intersection tests. For example, for inspecting a ray segment this optimization pays off.

So far, we have focussed on a very flexible hierarchy that is build around the  $B_i$ . If we assume the neighborhoods are retained during deformation, we can instead build the hierarchy based on the bounding volumes enclosing the surface and pre-compute the contributing point samples. This results in a tighter acceleration structure which increases performance during intersecting the surface with rays. In addition, the surface can be evaluated more efficiently, as only the leaf node containing the query point has to be found.

This optimization has been applied by Adams et al. [1] in the setting of physics-based simulation of elastically and even plastically deforming solids. They compute neighborhoods once and re-use them for the deformed surface in subsequent frames. They apply this for the surfaces resulting from a simulation framework presented by Müller et al [76]. Here, the sampling of the simulation domain is de-coupled from the sampling of the boundary surfaces which allows efficient animation of highly detailed models using a smooth displacement field  $\mathbf{u}$ .

The deformation field is defined at a relatively small number of simulation nodes. These nodes are discrete point samples of the volume of the model and the displacements from their original position completely define the deformation of the object's surface. As the number of surface samples is usually much higher than the number of simulation nodes (usually up to two or three orders of magnitudes), updating the bounding sphere hierarchy from the simulation nodes is significantly faster than updating it from the surface points.

The implicit spatial coherence can be exploited for efficient updates of the bounding sphere hierarchy. This idea of dynamically updating a bounding sphere hierarchy from a deformation field was proposed by James and Pai [59] in the context of collision detection. For each sphere bounding a sample, a list of the simulation nodes is stored that define the displacement of the samples. The updated sphere center and radius will be computed from the displacement of these simulation nodes.

The optimizations can be generalized to fit in any animation method that applies the idea of an embedded surface (such as for example the point-based shell animation framework of Wicke et al. [102]).

Instead of recomputing the bounding sphere hierarchy from scratch when the object deforms, Adams et al. propose to update the existing hierarchy from the deformation field. Given the list of simulation nodes associated to each sphere, the sphere  $(\mathbf{x}_c, R)$  can be updated to bound the deformed surface as follows.

The new radius  $R'$  is conservatively estimated from the maximal distance between the deformed points and the new locations. The radius update is always done with respect to the initial (optimal) bounding spheres, i.e., the radius can both increase and decrease over time. The sphere hierarchy thus maintains its tight fit even for highly elastic materials that expand and shrink significantly during an animation.



# Chapter 6

## Bounded Manifolds and Non-orientable Surfaces

Most point-based approaches seem to assume that the represented surface is a solid and, thus, unbounded and globally orientable. This might stem from the fact that point clouds are mostly acquired by scanning real-world (solid) objects, so that areas lacking points are considered inadequately sampled and to be fixed. However, with our interest in unstructured point sets as a general surface representation, we feel it is important to consider surfaces with boundary explicitly. Furthermore, enhancing reconstruction to faithfully deal with bounded, possibly non-orientable surfaces and manifolds in general increases the robustness against sampling errors and makes the algorithm more versatile.

We extend the basic definition of curves and surfaces (as introduced in Chapter 2) to handle smooth boundaries and show that surfaces could be (globally) non-orientable. In this chapter we make the following contributions:

**Boundary definition:** We enhance the Manifold definition and define a smooth and natural boundary only requiring the quantities we compute during the basic manifold approximation.

**Boundary behavior and parameters:** We prove several properties about the behavior of the computations involved in the estimation of the boundary. We use these approximations to solve the problem of parameter estimation and provide the user with an interval of reasonable parameters to choose from.

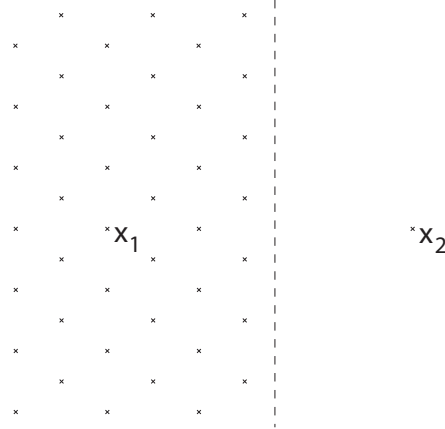
**Orientation:** For surfaces, the computation procedure requires to approximate planes that locally fit the surface. As the orientation of the normals defining these planes is irrelevant, a global orientation of normals is not required. We show that the smoothness of the surface is not affected.

Despite these features, the point set could still define a solid and our implicit definition of the surface would allow defining inside and outside.

In the following, we the definition of the boundary and how to choose the necessary parameters. We, restrict ourselves to the setting of weighting functions with equal radii for all  $\mathbf{p}_i$  because our reasoning concerning the parameters is based on the simpler assumption. Nevertheless, the approach we present can also be used in more general settings.

## 6.1 Defining the boundary

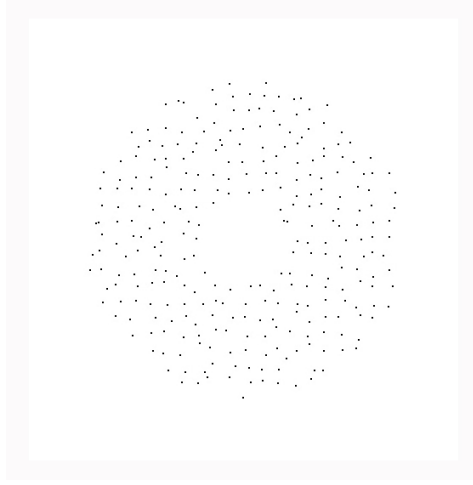
For defining a boundary, we illustrate our reasoning using Figure 6.1: The input points are distributed equally over a half-plane. The two points of evaluation  $\mathbf{x}_1$  and  $\mathbf{x}_2$  as well as all points in  $\mathcal{P}$  are assumed to be co-planar. The direction of smallest weighted co-variance of  $\mathbf{x}_1$  and  $\mathbf{x}_2$  is, thus, the constant normal to the plane. Thus,  $\mathbf{x}_1$  is part of the surface, and  $\mathbf{x}_2$  is part of the surface if  $\mathbf{x}_2 \in \Omega$ .



**Figure 6.1:** The rationale of defining bounded surfaces: While all points are in the same plane, we expect  $\mathbf{x}_2$  not to be part of the reconstructed surface.

The surface is defined only inside the neighborhood  $\Omega$  of the input points, which is bounded when using weight functions that are compactly supported. Using  $\partial\Omega$  for the definition of the boundary is not a good idea in practice: the computations involved in evaluating Equations 2.5-2.7 break down near the boundary of  $\Omega$  because the sum of weights in the denominators tends to zero (cf. Equations 2.1 and 2.3). For illustration purposes, we will use an irregular point set representing a planar disk with a hole (see





**Figure 6.2:** The ring model: A set of points representing a planar disk with a hole.

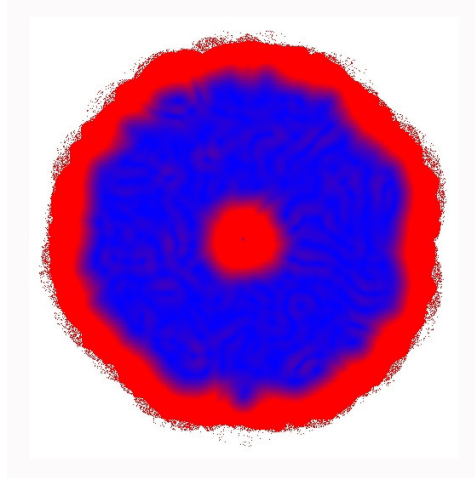
Figure 6.2). The numerical breakdown of our implementation close to  $\partial\Omega$  can be clearly observed in Figure 6.3, showing a rendering of the surface generated by ray casting.

The numerical problem could be avoided by requiring the sum of weights to exceed a certain threshold, i.e.  $\sum_i \theta(\|\mathbf{p}_i - \mathbf{x}\|) > \epsilon$ . This might lead to a definition of boundaries that are robustly computed, however, we feel that this is a bad idea: The boundary would depend on the sampling density. To illustrate this, imagine any point set representing a surface with holes. A given threshold on the sum of weights defines the boundary. Note that this boundary could be arbitrarily shifted towards  $\partial\Omega$  by simply adding points in the locations of the existing points.

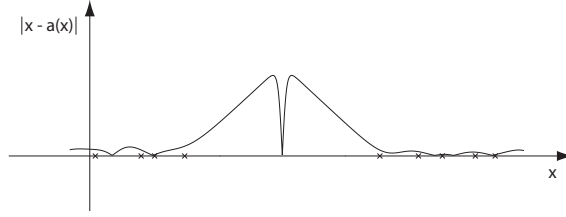
In order to efficiently compute the implicit function  $f$  an enclosing ball structure  $\mathcal{B}$  representing  $\Omega$  can be maintained (cf. 5). The balls then have a radius  $r_B < r_{\theta_i}$ , the radius of support for  $\theta_i$ . For adequate  $r_B$  all points in the set of enclosing spheres could be robustly evaluated and the numerical breakdown close to  $\partial\Omega$  is avoided. One might use  $\partial\mathcal{B}$  to define the boundary of  $\mathcal{S}$ , however, this results in a piecewise circular boundary that is only  $C^0$  where circular arcs meet.

We propose a different approach to obtain boundaries that are smooth, easy to compute, and almost insensitive to variations in the sampling density. Refer again to Figure 6.1: To distinguish the location of  $\mathbf{x}_1$  and  $\mathbf{x}_2$  we inspect the relative location of the local centroid  $\mathbf{c}$  of the points. Note, that the local centroid is necessarily contained in the convex hull of the contributing points. Thus, for points far away from the point set the distance  $\|\mathbf{x} - \mathbf{c}(\mathbf{x})\|$  increases, while we expect this distance to be rather small for locations close to (or “inside”) the points. For further use we will call

$$o(\mathbf{x}) = \|\mathbf{x} - \mathbf{c}(\mathbf{x})\| \quad (6.1)$$



**Figure 6.3:** A rendering resulting from ray casting the points shown in Fig. 6.2. The support radius of the weight functions  $r_\theta$  is chosen relatively small; the breakdown of the evaluation of points close to  $\partial\Omega$  can be clearly observed.



**Figure 6.4:** A plot of  $c(\mathbf{x}) = \|\mathbf{a}(\mathbf{x}) - \mathbf{x}\|$  for a slightly noisy point set with a gap.

the *off-center value* of  $\mathbf{x}$ .

The main idea for defining a boundary is to require the off-center value to be less than a user-specified threshold. More precisely, the definition of manifold  $\mathcal{M}_d$  in Equation 2.8 becomes

$$\mathcal{M}_d = \{\mathbf{x} \in \Omega \mid f(\mathbf{x}) = 0 \wedge c(\mathbf{x}) < \epsilon_o\}. \quad (6.2)$$

In the following section we will discuss reasonable choices for  $\epsilon_o$ , the *off-center limit*.

## 6.2 Choosing the off-center limit

For the following discussion we use a series of plots of  $o(\mathbf{x})$  in one dimension and corresponding renderings produced with ray casting and color coding  $o(\mathbf{x})$  of the disk point set shown in Figure 6.2. The input points in one dimension could be interpreted as a one-dimensional slice through the disk point set: The points result from regularly sampling the line, adding Gaussian noise to the locations, and removing some points to generate a gap. We assume the space inside the gap is in  $\Omega$ , i.e.  $r_\theta$  is sufficiently large. The graph of  $o(\mathbf{x})$  in one dimension is shown in Figure 6.4.

As expected, the off-center value  $o(\mathbf{x})$  is small in-between the points (in fact, for equidistant input points far away from the boundary of  $\Omega$  it is  $o(\mathbf{x}) = 0$ ) and it increases almost linearly far away from the points. This is also reflected in the color coded renderings of the disk (see Figures 6.3, 6.5, and 6.6), where blue indicates small values and red indicates large values of  $o(\mathbf{x})$ . Note, that the off-center value is particularly small in the barycenter of points – including the barycenter of holes. Figure 6.6 shows what this behavior entails if the radius of the sphere structure  $r_B$  is relatively large: Inside the hole a small island appears. Other potential problems are

- holes in areas of the surface that seem to be adequately sampled. This results from a value for  $\epsilon_o$  that is too small (see Fig. 6.5a),
- piecewise circular boundaries resulting from a too large value for  $\epsilon_o$  (see Fig. 6.5b).

It is clear that we have to define  $\epsilon_o$  relative to  $r_B$ . Before we can do so we need to establish a connection between  $o(\mathbf{x})$  and the distance of  $\mathbf{x}$  to the point set  $d_{\mathcal{P}}(\mathbf{x})$ . We can show (see Section 6.4) that

1.  $o(\mathbf{x}) > \sqrt{d_{\mathcal{P}}^2(\mathbf{x}) - h^2(2 - \sqrt{3})}$  moves away from the points and
2.  $\lim_{d_{\mathcal{P}}(\mathbf{x}) \rightarrow \infty} d_{\mathcal{P}}(\mathbf{x}) = o(\mathbf{x})$  (assuming that  $\theta$  has very large radius of support).

In other words,  $o(\mathbf{x})$  behaves roughly like the distance to the point set, however, is larger when  $\mathbf{x}$  moves away from the surface (a notable exception is around the barycenter of holes, where  $d_{\mathcal{P}}$  is maximal, while  $o(\mathbf{x})$  is small).

For an exact characterization of  $\epsilon_o$  we consider two different cases for the radius  $r_B$  of the enclosing ball structure. First, consider  $r_B$  to be chosen in a natural way, i.e. so that a portion of the hole/gap is not covered by  $\mathcal{B}$  (Fig. 6.5). In order to avoid unwanted holes in the surface  $\epsilon_o$  needs to be chosen large enough, however,  $\partial\mathcal{B}$  defines an upper bound on  $\epsilon_o$ : Because  $o(\mathbf{x}) > \sqrt{d_{\mathcal{P}}^2(\mathbf{x}) - h^2(2 - \sqrt{3})}$  and  $r_B$  reflects a bound on  $d_{\mathcal{P}}$  we require  $\epsilon_o < \sqrt{r_B^2 - h^2(2 - \sqrt{3})}$ .

To bound  $\epsilon_o$  from below, we need to find a reasonable maximum of  $o(\mathbf{x})$  in cases where we would consider the surface closed. In our setting, the only reasonable definition

of a closed surface is to assume it is closed where it is contained in  $\mathcal{B}$ . Thus, the case where the enclosing ball structure just covers a hole is the relevant limiting case (see Fig. 6.6 for an illustration). In order to avoid the island (which could as well be seen as an unwanted hole in the surface),  $\epsilon_o$  has to be larger than the maximum of  $o(\mathbf{x})$  in this region.

Note, that because  $o(\mathbf{x})$  is larger than  $d_{\mathcal{P}}(\mathbf{x})$  as we move away from the boundary of a hole, it is **not** necessary that an  $\epsilon_o < r_B$  that is larger than  $o(\mathbf{x})$  in small gaps (i.e. those that are covered by  $\mathcal{B}$ ) exists. To show that such an  $\epsilon_o$  exists in practice, we need to consider the practical choice of  $r_B$  based on  $h$ , and the way we determine  $h$  from the distribution of the points. In particular, by assuming Gaussian weights of the form  $\theta(d) = e^{-d^2/h^2}$  we estimate  $o(\mathbf{x})$  to be bounded by  $h$  (see section 6.5). Given our practical choice of  $r_B$  this leads to the following bounds for  $\epsilon_o$ :

$$\frac{1 + 4\sqrt{3}}{9}r_B > \epsilon_o > \frac{2}{3}r_B \quad (6.3)$$

Figures 6.5c and 6.6b show reasonable values for  $\epsilon_o$  for both setting of radii  $r_B$ .

## 6.3 Results

We have tested our approach by ray casting several surfaces represented by unstructured point sets using the ray intersection method described in Section 3. The point sets all represent surfaces with boundaries, and some are globally non-orientable.

In all our examples we have estimated the feature size  $h$  by finding the 6 nearest neighbors for all points and then computing the mean distance. The radius  $r_B$  of the enclosing sphere structure is set to  $1.5h$ . The limit for the off-center value is  $\epsilon_c = 0.75r_B$  in all examples. Note that we have not yet encountered the problem of islands for these values in practice.

The Stanford Bunny model is known to have several holes on its bottom side. When we render the original point set only the two large holes appear, while the two longish holes are closed. We have up-sampled the original data set using the methods described in [9] so that we can visualize all four holes. The result is depicted in Figure 6.7. The color coded image reveals that  $o(\mathbf{x})$  is large in areas of small sampling density but also in areas of high curvature. This could be only due to poor normal estimation, as the local centroid is rather expected to lie inside (resp. outside) of the surface.

To demonstrate that the surface could as well be non-orientable we have rendered a Möbius strip and a Klein bottle represented by points (see Figure 6.8). The point set for the Möbius strip consists of 2.6k points, and the Klein bottle of 33k points. We have cut out a hole in the Klein bottle so that the surface is free of self-intersections. We have used a semi-transparent (non-refracting) material to show also the inner part of the shapes.

## 6.4 Asymptotic relation of distance to point set and off-center value

We are interested in the behavior of  $o(\mathbf{x}) = \|\mathbf{c}(\mathbf{x}) - \mathbf{x}\|$  relative to  $d_{\mathcal{P}}(\mathbf{x})$ . As all quantities are scalars it is sufficient to inspect the problem in one dimension. W.l.o.g. we can sort and translate the points so that  $p_0 = 0$ ,  $p_1 = -\delta$ ,  $\delta > 0$ , and  $p_i \leq -\delta$ ,  $i > 1$ . Then

$$o(x) = \left| \frac{p_0\theta(|p_0 - x|) + \sum_{i>0} p_i\theta(x + p_i)}{\theta(|p_0 - x|) + \sum_{i>0} \theta(x + p_i)} - x \right| \quad (6.4)$$

and by exploiting the properties of the  $p_i$  this can be re-written as

$$o(x) = \left| \frac{-\sum_{i>0} |p_i|\theta(x + p_i)}{\theta(|x|) + \sum_{i>0} \theta(x + p_i)} - x \right|. \quad (6.5)$$

Because we are interested in the behavior of  $o(x)$  as  $x$  moves away from the points we restrict our analysis to  $x > 0$ , which allows writing

$$o(x) = \left| -\frac{\sum_{i>0} |p_i|\theta(x + p_i)}{\theta(x) + \sum_{i>0} \theta(x + p_i)} - x \right| \quad (6.6)$$

and using that  $\theta$  is defined to be strictly positive in  $\Omega$  this is equivalent to

$$o(x) = \frac{\sum_{i>0} |p_i|\theta(x + p_i)}{\theta(x) + \sum_{i>0} \theta(x + p_i)} + x. \quad (6.7)$$

For our particular choice of  $\mathcal{P}$  we have  $d_{\mathcal{P}}(x) = x$ , which immediately shows  $o(x) > d_{\mathcal{P}}(x)$ . We claim further that the asymptotic behavior for  $x \rightarrow \infty$  is  $o(x) \approx d_{\mathcal{P}}(x)$ . Since  $\sum_{i>1} |p_i|\theta(x + p_i) < m \sum_{i>1} \theta(x + p_i)$  for an appropriate  $m < \infty$ , it is sufficient to show that

$$\lim_{x \rightarrow \infty} \theta(x) = \theta(x + \delta), \quad (6.8)$$

which follows directly from the requirement of monotone decreasing behavior for  $\theta$  (and also holds for  $\delta = 0$ ).

For the practical case, we assume the points to be scattered on a two-dimensional manifold (with boundary). Because  $h$  is determined as the average distance between close points, we assume that on the manifold the (geodesic) distance to any of the sample points is less than  $h/2$ . We establish an orthonormal coordinate system  $x, y, z$  at the boundary of the manifold, so that  $z$  points in normal direction and  $x$  is orthogonal to the boundary. If the manifold was planar then the distance  $d_{\mathcal{P}}$  of points on the  $x$ -axis was simply bounded by  $\sqrt{x^2 + h^2/4}$ . Since we assume the manifold cannot represent features smaller than  $h$ , the radius of curvature is larger than  $h$ . Thus, the maximum deviation

in normal direction between two points at a distance  $h/2$  is less than  $h - \sqrt{h^2 - h^2/4}$ . This leads to

$$d_{\mathcal{P}} = \sqrt{x^2 + h^2/4 + \left(h - \sqrt{h^2 - h^2/4}\right)}$$

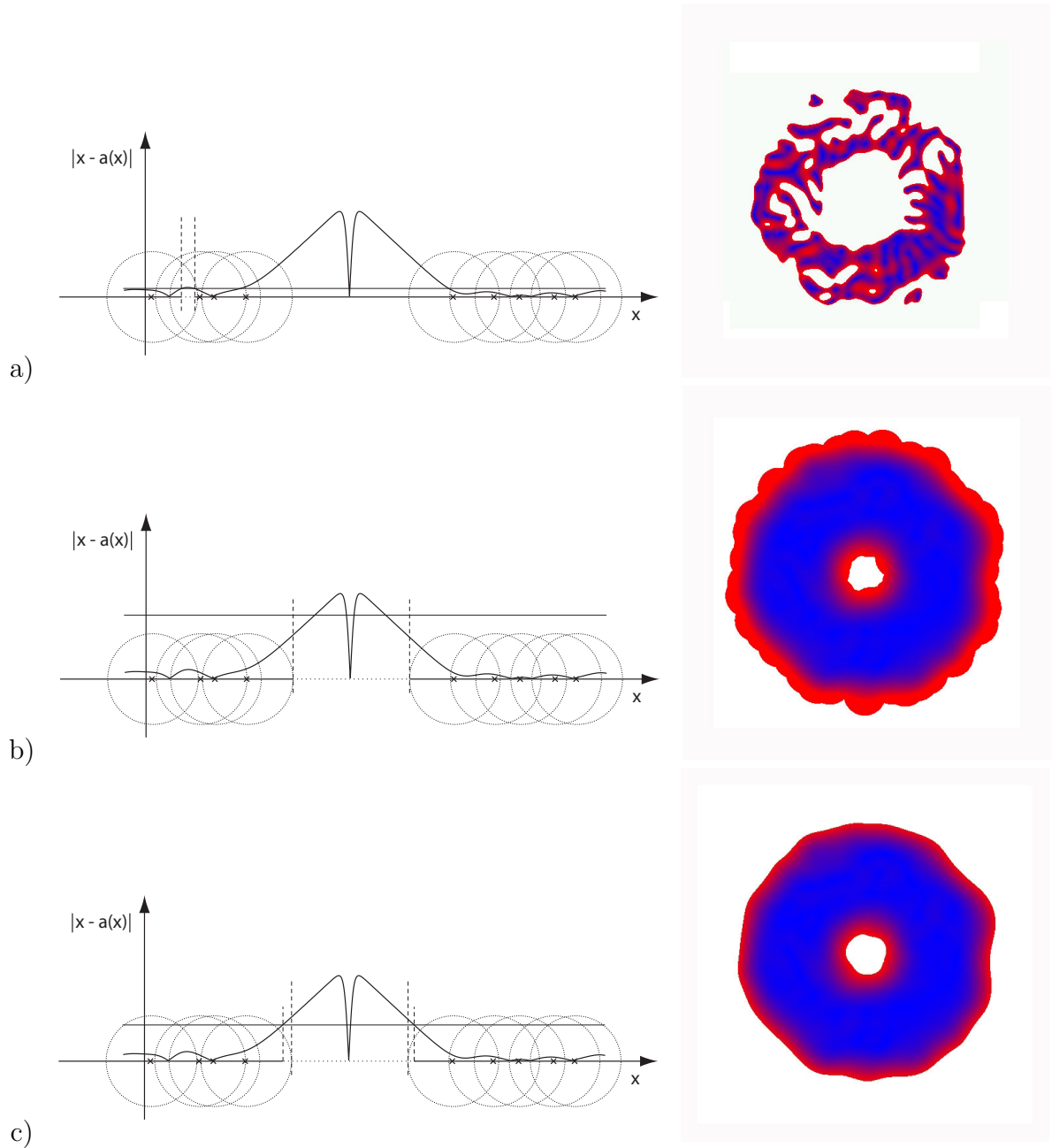
for points on the  $x$ -axis an we conclude

$$d_{\mathcal{P}} = \sqrt{d_{\mathcal{P}}^2 - h^2 \left(2 - \sqrt{3}\right)}$$

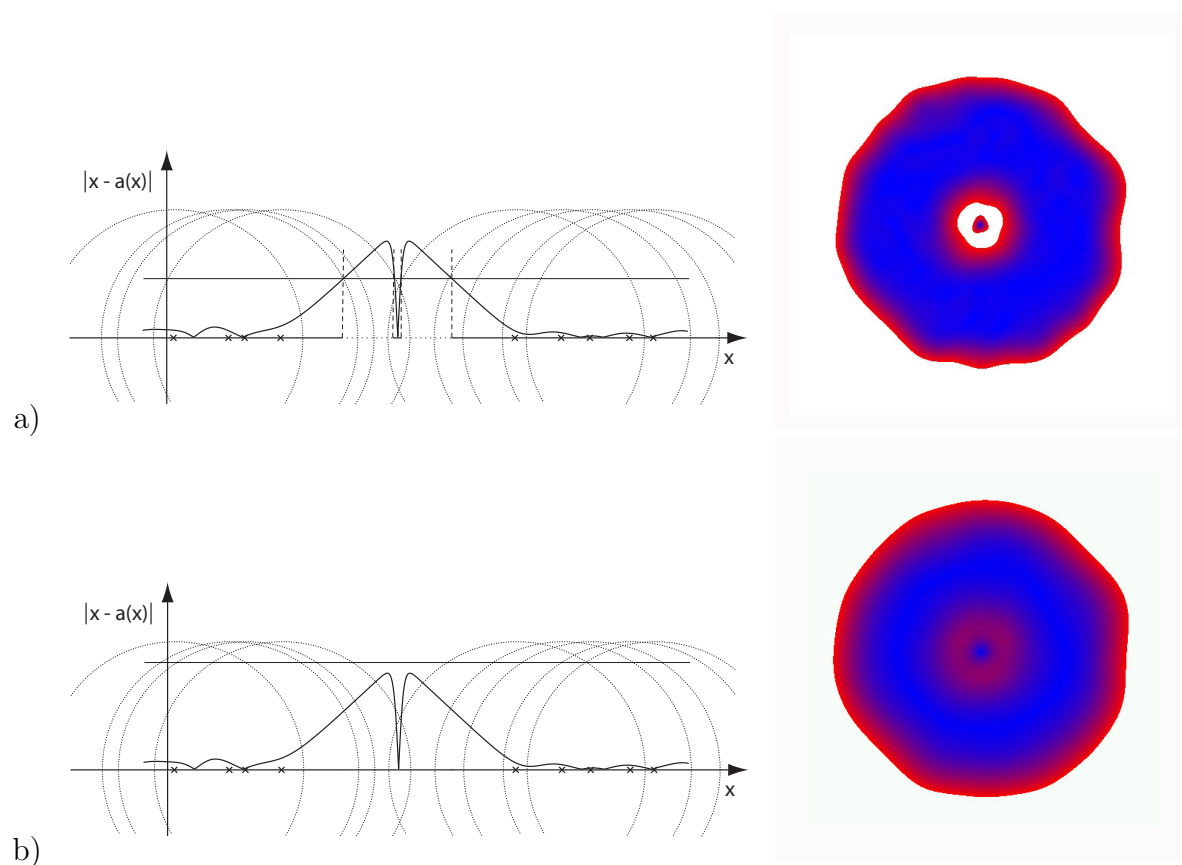
The asymptotic behavior for large distances remains unchanged because we assume  $d_{\mathcal{P}}(x) \gg h$ .

## 6.5 Bounds on the off-center value in small gaps

We consider sets of points on the real line with spacing roughly  $h$  containing the points  $-1.5h$  and  $+1.5h$  but no points in the interval  $] -1.5h, 1.5h[$ , representing a gap of  $2r_B$  (assuming our preferred choice of  $r_B = 1.5h$ ). We have performed a large number of numerical experiments to evaluate the maximum of  $o(x)$ . Keeping the points at the boundary of the gap and allowing an arbitrary amount of noise for the other points we find that  $o(x) < h$  under all circumstances. Note that we could compute the maximum of  $o(x)$  analytically for special settings, however, we feel more comfortable with an exhaustive numerical experiment as the special settings might not cover all real-world cases.

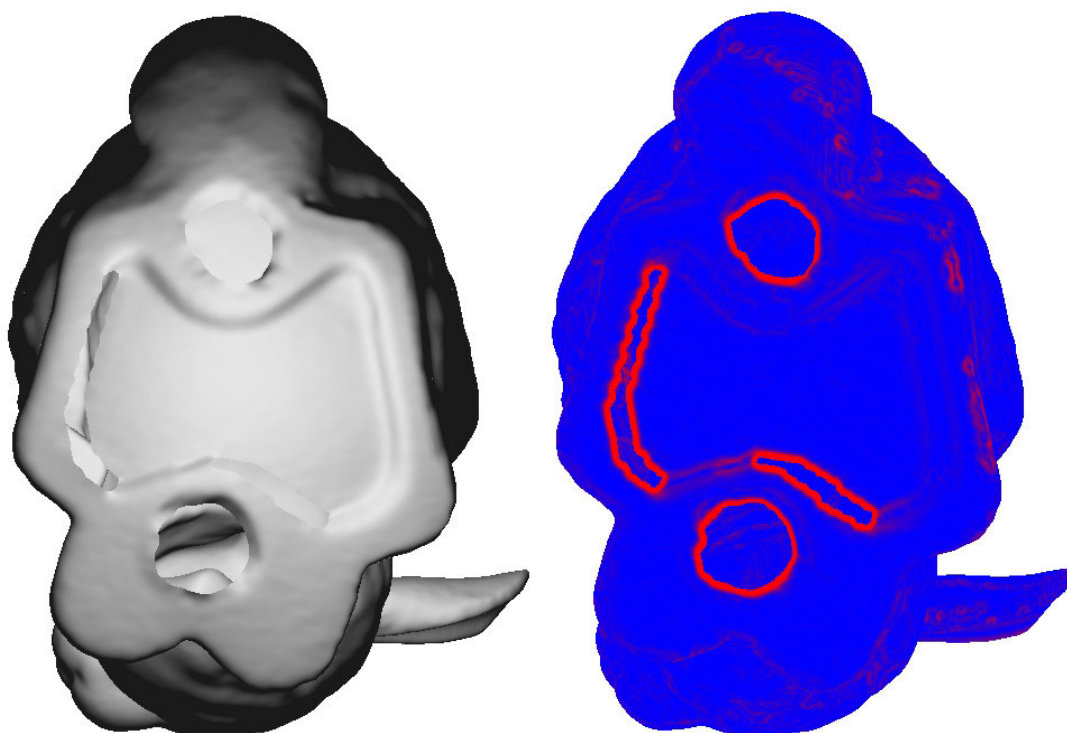


**Figure 6.5:** Different choices for the off-center value given a reasonable setting for the radius of the enclosing ball structure. In (a) the off-center value is chosen too small and holes appear in the surface where it is expected to be connected. In (b) the off-center value is too large so that surface is bounded only by the enclosing sphere structure. The off-center value in (c) shows a reasonable choice.

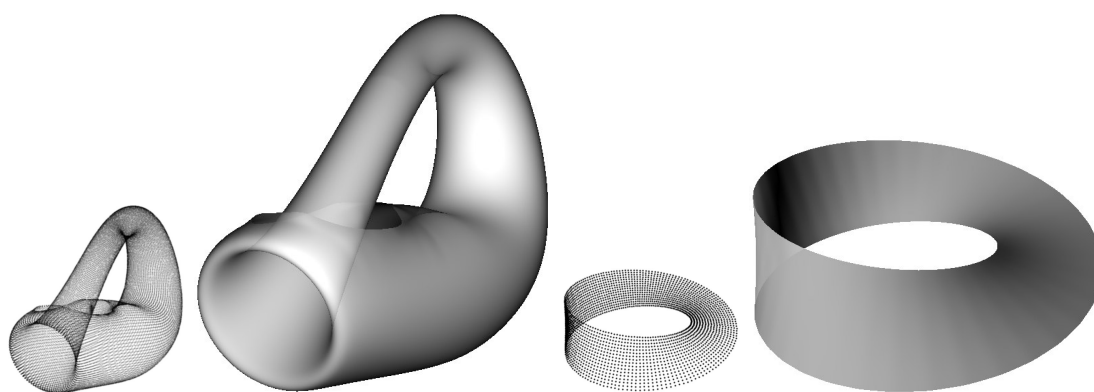


**Figure 6.6:** The enclosing ball structure just covers the hole in the point set. If the off-center value is too small (a), an island appears, however, we can estimate a sufficient condition for practical cases (b) so that no islands appear. Note that this entails that surface is closed wherever it is contained in the enclosing sphere structure.





**Figure 6.7:** Renderings of the Stanford bunny model showing the holes in the bottom. The first image shows a rendering resulting from ray casting including shadow rays. The second image is color coded depicting the off-center value ( $\|\mathbf{x} - \mathbf{a}(\mathbf{x})\|$ ).



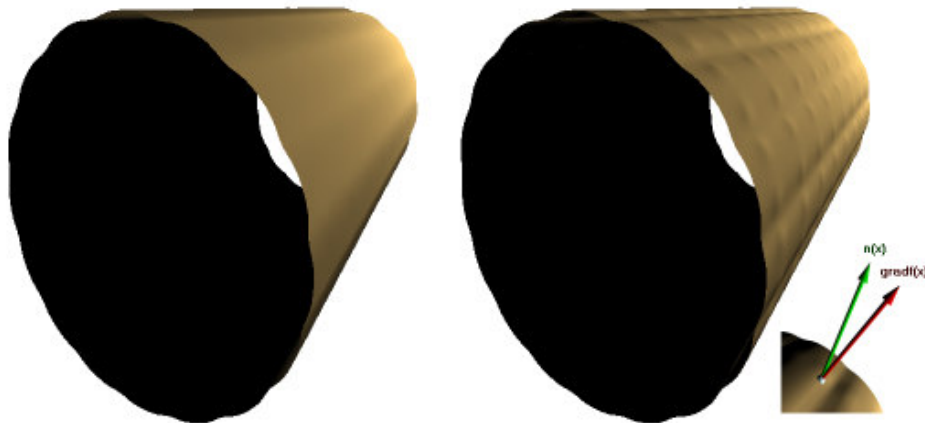
**Figure 6.8:** Renderings of non-orientable, bounded surfaces represented by unstructured point sets. Note, that we have added a boundary to the Klein bottle where it would have self-intersected.



# Chapter 7

## Differential Geometry on Surfaces

In many applications, it is not sufficient to know a surface. Higher order information, such as the normal and the curvature on the surface are also of interest. For instance in rendering, the normal has to be known in order to simulate the interaction of light with the material. The orthogonal projection operator presented in Section 3.1.3 requires to evaluate the exact surface normal. Curvature provides information about the surface. For instance, optimized meshes can be produced by following the directions of principle curvature [11]. There is a variety of other applications including where normal and curvature information is of interest. This includes particle simulations, flow simulations, animation and pattern matching. In comparison to the MLS surface definition (refer to Section 1.2.4), our implicit description allows the exact evaluation of surface normals, i.e. without taking finite differences.



**Figure 7.1:** Comparison of  $n(x)$  (left) and  $\nabla f(x)$  (right) computed for a very small value  $h$ . For larger values  $h$  the difference becomes less obvious, as for the smoother surface  $n(x)$  becomes a better approximation of  $\nabla f(x)$ .

So far, we have simply used the local tangent frame  $\mathbf{T}_x$ . It's orthogonal direction is a good approximation of the surface normal, however it is not equivalent. When computed this way, some of the surface details are not visible. In order to determine exact surface normals, we have to compute the gradient of the implicit function  $f$ , which involves differential geometry. The difference can be seen in figure 7.1.

Other representations, such as triangle meshes do neither provide exact normals nor curvature. To smooth the piecewise linear surface, normals are usually computed by first averaging the plane normals adjacent to the vertices and then by interpolating these linearly. On triangle meshes curvature is either zero on the faces and infinitely high on the edges. To introduce useful curvature, a region around a vertex can be considered. The eigen-elements of this tensor yield an estimation of the curvature (discrete differential geometry). The quality of the estimation highly depends on how fine the tessellation is. For unstructured representations such as points clouds the curvature is often estimated by fitting a polynomial approximation (or quadric) and considering the curvature of that smooth surface.

Following we describe, how to obtain the exact gradient (Section 7.1) and curvature tensor (Section 7.2) for our surface  $\mathcal{S}$  which is implicitly defined according to Equation 2.9.

## 7.1 Surface Normal

As we will see further below, the orthogonal direction  $\mathbf{n}_0(\mathbf{x})$  is not orthogonal to the surface  $\mathcal{S}$ , although it used to define the implicit function  $f$ . In Chapter 2.5 different variants of the surface definition have been proposed. For the following calculations we limit the scope to the cases where  $\mathbf{n}_0(\mathbf{x})$  is defined as the local centroid and the direction of smallest covariance. For simplicity we set  $\mathbf{n}(\mathbf{x}) = \mathbf{n}_0(\mathbf{x})$ .

We examine the gradient of  $f$  in the ortho-normal system  $\{\mathbf{e}_k\}$ , i.e.

$$\nabla f(\mathbf{x}) = \left( \frac{\partial f(\mathbf{x})}{\partial \mathbf{e}_0}, \frac{\partial f(\mathbf{x})}{\partial \mathbf{e}_1}, \dots \right). \quad (7.1)$$

The product rule for differentiating vector fields yields the directional derivatives of  $f$ :

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{e}_k} = \frac{\partial \mathbf{n}(\mathbf{x})^\top}{\partial \mathbf{e}_k} (\mathbf{x} - \mathbf{c}(\mathbf{x})) + \mathbf{n}(\mathbf{x})^\top \left( \mathbf{e}_k - \frac{\partial \mathbf{c}(\mathbf{x})}{\partial \mathbf{e}_k} \right), \quad (7.2)$$

We see that the difference between  $\nabla f(\mathbf{x})$  and the normal direction  $\mathbf{n}(\mathbf{x})$  is not necessarily in the direction of  $\mathbf{n}(\mathbf{x})$ . Computing the gradient requires the evaluation of directional derivatives of  $\mathbf{n}(\mathbf{x})$  and  $\mathbf{c}(\mathbf{x})$ .

**Computing  $\frac{\partial \mathbf{c}}{\partial \mathbf{e}_k}$ .** Taking directional derivatives of  $\mathbf{c}(\mathbf{x})$  along the basis directions  $\mathbf{e}_k$  is straightforward and yields

$$\frac{\partial \mathbf{c}(\mathbf{x})}{\partial \mathbf{e}_k} = \frac{(\sum_i \theta_i(\mathbf{x})) \left( \sum_i \frac{\partial \theta_i(\mathbf{x})}{\partial \mathbf{e}_k} \mathbf{p}_i \right) - \left( \sum_i \frac{\partial \theta_i(\mathbf{x})}{\partial \mathbf{e}_k} \right) (\sum_i \theta_i(\mathbf{x}) \mathbf{p}_i)}{(\sum_i \theta_i(\mathbf{x}))^2}, \quad (7.3)$$

where  $\theta_i = \theta(\|\mathbf{x} - \mathbf{p}_i\|)$  and  $\theta'_i = \theta'(\|\mathbf{x} - \mathbf{p}_i\|)$ .

**Computing  $\frac{\partial \theta(\mathbf{x})}{\partial \mathbf{e}_k}$ .** We also have to compute the partial derivatives of the weighting functions  $\theta_i(\mathbf{x}) = \theta(\frac{\|\mathbf{x} - \mathbf{p}_i\|}{r_i})$ , where  $r_i$  is the radius of support. Under the assumption that  $\theta'(x)$  is given, the partial derivatives of the weighting functions are:

$$\frac{\partial \theta_i(\mathbf{x})}{\partial \mathbf{e}_k} = \frac{\mathbf{e}_k^\top (\mathbf{x} - \mathbf{p}_i)}{r_i \|\mathbf{x} - \mathbf{p}_i\|} \theta' \left( \frac{\|\mathbf{x} - \mathbf{p}_i\|}{r_i} \right). \quad (7.4)$$

These partial derivatives have a singularity at  $\mathbf{p}_i$ : if  $\mathbf{x} = \mathbf{p}_i$ , then  $\|\mathbf{x} - \mathbf{p}_i\|$  is zero and this would result in a division by zero. In that rare case we just set  $\frac{\partial \theta_i(\mathbf{x})}{\partial \mathbf{e}_k} = 0$ , which works very well. Note that we assume the derivative of the weight functions can be computed analytically, which is certainly true for the typically used piecewise polynomial functions.

**Computing  $\frac{\partial \mathbf{n}(\mathbf{x})}{\partial \mathbf{e}_k}$ .** If  $\mathbf{n}(\mathbf{x})$  is given as the local centroid of normals, the partial derivatives of

$$\frac{\partial \mathbf{n}(\mathbf{x})}{\partial x_k} = \frac{\partial \frac{\mathbf{m}(\mathbf{x})}{\|\mathbf{m}(\mathbf{x})\|}}{\partial x_k} = \frac{\frac{\partial \mathbf{m}(\mathbf{x})}{\partial x_k}}{\|\mathbf{m}(\mathbf{x})\|} - \frac{\frac{\partial \mathbf{m}(\mathbf{x})}{\partial x_k}^\top \mathbf{m}(\mathbf{x})}{\|\mathbf{m}(\mathbf{x})\|^3} \mathbf{m}(\mathbf{x}) \quad (7.5)$$

where

$$\mathbf{m}(\mathbf{x}) = \sum_i w_i(\mathbf{x}) \mathbf{n}_i$$

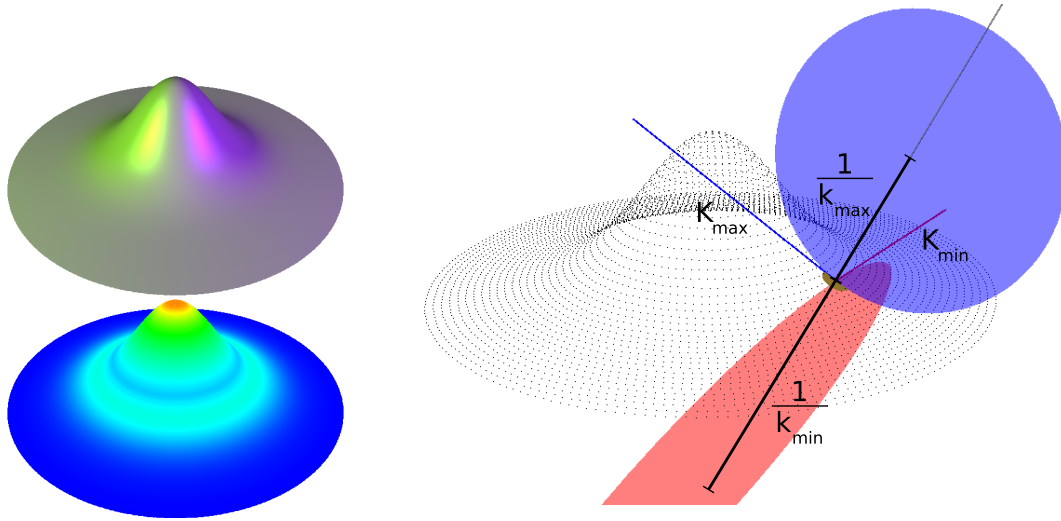
and its partial derivatives are

$$\frac{\partial \mathbf{m}(\mathbf{x})}{\partial x_k} = \sum_i \frac{\partial w_i(\mathbf{x})}{\partial x_k} \mathbf{n}_i.$$

If  $\mathbf{n}(\mathbf{x})$  is given as the direction of smallest covariance, the computation becomes slightly more complicated. We use closed forms for calculating the eigenvalues of the covariance matrix  $\mathbf{C}$  (from Equation 2.14), which are available for the most relevant 2D and 3D case. For details refer to [30].

## 7.2 Surface Curvature

For a point  $\mathbf{x}$  on a surface in  $\mathbb{R}^3$ , consider the plane that contains the normal vector and a tangent vector at that point. The intersection between the surface and that plane is a plane curve and has a curvature. The absolute value of the curvature of the plane curve at  $\mathbf{x}$  is the inverse of the radius of the osculating circle of the plane curve at  $\mathbf{x}$ , the sign of the curvature value denotes in which direction the curve turns relative to the normal vector. This curvature value varies with the chosen tangent vector, the minimum and the maximum are called the principal curvatures  $k_{min}$  and  $k_{max}$  at  $\mathbf{x}$ , the corresponding tangent vectors are called the principle curvature directions  $\mathbf{K}_{min}$  and  $\mathbf{K}_{max}$ . Note that  $\mathbf{K}_{min}$  and  $\mathbf{K}_{max}$  always are perpendicular (the only exception is the case  $k_{min} = k_{max}$ , in that case  $\mathbf{K}_{min}$  and  $\mathbf{K}_{max}$  can be chosen arbitrarily from the tangent vectors). Figure 7.2 illustrates this for a surface that represents a 2-d Gaussian function.



**Figure 7.2:** Illustration of surface curvature. Left, top: A surface representing a 2-d Gaussian function. Right: A visualization of the osculating circles for the minimum (red) and maximum (blue) principal curvatures at one point of the surface. Left, bottom: A color coded visualization of the maximum principal curvature for the whole surface (red means high curvature, blue means low curvature).

There is a variety of different scalar measures for curvature[34]:

- Gaussian Curvature  $k_{min} * k_{max}$
- Mean Curvature  $\frac{k_{min} + k_{max}}{2}$
- Maximal absolute Curvature  $\max(|k_{min}|, |k_{max}|)$

- Shape Index  $-\frac{2}{\pi} \arctan \frac{k_{max}+k_{min}}{k_{max}-k_{min}}$

To compute the curvature, it is necessary to first compute the Hessian matrix of the implicit function  $f(\mathbf{x})$ :

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_3} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_3} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_3 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_3 \partial x_2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_3 \partial x_3} \end{pmatrix} = \begin{pmatrix} \frac{\partial \nabla f(\mathbf{x})}{\partial x_1} & \frac{\partial \nabla f(\mathbf{x})}{\partial x_2} & \frac{\partial \nabla f(\mathbf{x})}{\partial x_3} \end{pmatrix}.$$



**Figure 7.3:** From left to right: rendering using the normal average, rendering using the exact surface normal and the absolute maximum curvature color coded.

The formulas to compute  $\mathbf{H}$  are not particularly difficult, but they are quite long, so they are omitted here (for details cf [30]). Under the assumption that  $\theta''(x)$  is given the second order partial derivatives of the weighting functions are:

$$\frac{\partial^2 w_i(\mathbf{x})}{\partial x_k \partial x_l} = \begin{cases} \frac{\mathbf{e}_k^T(\mathbf{x}-\mathbf{p}_i) \cdot \mathbf{e}_l^T(\mathbf{x}-\mathbf{p}_i)}{r_i \|\mathbf{x}-\mathbf{p}_i\|^2} \left( \frac{\theta''(\frac{\|\mathbf{x}-\mathbf{p}_i\|}{r_i})}{r_i} - \frac{\theta'(\frac{\|\mathbf{x}-\mathbf{p}_i\|}{r_i})}{\|\mathbf{x}-\mathbf{p}_i\|} \right) + \frac{\theta'(\frac{\|\mathbf{x}-\mathbf{p}_i\|}{r_i})}{r_i \|\mathbf{x}-\mathbf{p}_i\|} & k = l \\ \frac{\mathbf{e}_k^T(\mathbf{x}-\mathbf{p}_i) \cdot \mathbf{e}_l^T(\mathbf{x}-\mathbf{p}_i)}{r_i \|\mathbf{x}-\mathbf{p}_i\|^2} \left( \frac{\theta''(\frac{\|\mathbf{x}-\mathbf{p}_i\|}{r_i})}{r_i} - \frac{\theta'(\frac{\|\mathbf{x}-\mathbf{p}_i\|}{r_i})}{\|\mathbf{x}-\mathbf{p}_i\|} \right) & k \neq l \end{cases}$$

Like the first order partial derivatives, the second order partial derivatives also have a singularity at  $\mathbf{p}_i$ : if  $\mathbf{x} = \mathbf{p}_i$ , then  $\|\mathbf{x} - \mathbf{p}_i\|$  is zero and this would result in a division by zero. In that rare case we again just set  $\frac{\partial^2 w_i(\mathbf{x})}{\partial x_k \partial x_l} = 0$ , which works very well.

The Hessian matrix  $\mathbf{H}$  represents the curvature evolution of the scalar field  $f(\mathbf{x})$ . But to compute the curvature values and directions on the plane tangent to a point  $\mathbf{x}$  with  $f(\mathbf{x}) = 0$ , it is necessary to compute the matrix  $\mathbf{C}$  defined by the partial derivatives of the normalized gradient  $\mathbf{h}(\mathbf{x}) = \frac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|}$  (see [34]):

$$\mathbf{C} = \begin{pmatrix} \frac{\partial \mathbf{h}(\mathbf{x})}{\partial x_1} & \frac{\partial \mathbf{h}(\mathbf{x})}{\partial x_2} & \frac{\partial \mathbf{h}(\mathbf{x})}{\partial x_3} \end{pmatrix}.$$

The columns of  $\mathbf{C}$  can be computed using the columns of  $\mathbf{H}$ :

$$\frac{\partial \mathbf{h}(\mathbf{x})}{\partial x_k} = \frac{\partial \frac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|}}{\partial x_k} = \frac{\frac{\partial \nabla f(\mathbf{x})}{\partial x_k}}{\|\nabla f(\mathbf{x})\|} - \frac{\frac{\partial \nabla f(\mathbf{x})}{\partial x_k}^\top \nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|^3} \nabla f(\mathbf{x}).$$

One eigenvalue of  $\mathbf{C}$  has zero value and the corresponding eigenvector is collinear to  $\nabla f(\mathbf{x})$ , the other two eigenvalues are the principal curvatures  $k_{min}$  and  $k_{max}$  (with  $|k_{min}| \leq |k_{max}|$ ). The corresponding eigenvectors  $\mathbf{K}_{min}$  and  $\mathbf{K}_{max}$  are the principal curvature directions. Figure 7.3 shows several renderings of the afarensis model using approximated and exact surface normals and color coding the absolute maximum curvature.